klutzy.nanabi.org (http://klutzy.nanabi.org/)

MinGW & MinGW-w64 (./MinGW & MinGW-w64 @ klutzy.nanabi.org_files/MinGW & MinGW-w64 @ klutzy.nanabi.org.html) 2015-03-05

윈도에서 오픈 소스 프로그램을 써보려고 한다면 한번쯤은 MinGW라는 이름을 접하게 됩니다. 하지만 이게 어디에 쓰는 건지 홈페이지 설명을 봐서는 전혀 감이 오지 않을 거고요. 그저 "윈도에서도 gcc를 쓸 수 있다더라" 정도로 이해하는 게 보통일 겁니다.

MinGW는 Minimalist GNU for Windows의 약자로, 윈도 환경에서 GNU 툴을 이용하기 위한 최소한의 환경을 의미합니다. 이 환경에는 컴파일러 툴체인, 플랫폼 헤더 및 라이브러리가 포함됩니다. 여기에서 컴파일러 개발은 gcc나 binutils 저장소에서 이루어지며, MinGW는 그 gcc나 binutils를 빌드하고 실행하는 데에 필요한 나머지 부분을 구축하는 역할을 합니다.

MinGW는 Cygwin (http://www.cygwin.com/)과는 완전히 다른 타겟이라는 점을 유의해야 합니다. Cygwin은 윈도 위에 Linux 호환 레이어를 구현하는 프로젝트로, 추가 에뮬레이션 레이어가 들어갑니다. 따라서 Cygwin 위에서 돌아가는 프로그램은 Cygwin 환경이 필요하며, 다른 프로그램간의 상호작용 역시 Cygwin 레이어 위에서 동작합니다. MinGW는 그러한 레이어가 존재하지 않는 네이티브 윈도 타겟입니다.

MinGW

윈도에서 gcc를 이용해서 윈도 실행 파일을 만들고 싶다면 우선 컴파일러 툴체인이 윈도의 PE/COFF 아키텍처 (http://msdn.microsoft.com/en-us/gg463119.aspx)를 이해하고 있어야 합니다. 이건 gcc와 binutils에서 책임지고 있습니다.

윈도 API들의 함수 선언 헤더도 필요합니다. `windows.h`가 어딘가에 있어야 그 헤더를 쓰는 C 코드를 빌드할 수 있겠죠. 윈도 SDK를 설치하면 이들을 얻을 수 있지만, SDK가 오픈 소스가 아니라는 점이 여러 문제를 일으킵니다. MinGW의 w32api (http://sourceforge.net/projects/mingw/files/MinGW/Base/w32api/) 패키지에서 이 헤더들을 구현하고 있습니다. 이 파일들은 msdn 문서를 기반으로 손수 만들어지기 때문에 라이선스 문제에서 자유롭습니다. (초기에는 퍼블릭 도메인으로 배포되었고, MinGW 4.0부터 MIT 라이선스로 배포됩니다.) 물론 직접 만드는 것이다 보니까 불완전하거나 빠진 부분들이 종종 있고, 특히 윈도 최신 버전에 추가되는 API는 MinGW에 없는 경우가 많습니다.

C 표준 함수는 사정이 더 복잡합니다. MSVC++로 빌드한 바이너리는 MSVC++와 함께 제공되는 `msvcr###.dll` 런타임 라이브러리에 의존합니다. 이 라이브러리는 Visual C++ Redistributable Package로 설치할 수 있지만 오픈 소스가 아니며 재배포 제한이 존재합니다. 여기에서 GPL의 경우 더욱 문제가 될 수 있는 것이, GPL은 "시스템 라이브러리"와 링크할 수는 있지만 (http://www.gnu.org/licenses/gpl-faq.html#WindowsRuntimeAndGPL) 시스템 라이브러리를 함께 배포하는 것은 불가능하도록 명시합니다.

다만 MSVC++ 버전 6에서 사용하던 `msvcrt.dll`은 윈도 95 OSR 2.5부터 OS에 배포되고 있습니다 (https://support.microsoft.com/kb/175430/en-us/). 따라서 이 dll에는 배포 문제가 없습니다. C89만 지원한다는 (http://www.mingw.org/wiki/C99) 문제가 있지만요.. 따라서 MinGW의 C 런타임 라이브러리는 msvcrt.dll에 없는 함수들을 자체 구현합니다.

C++의 경우 C++ std 구현 라이브러리가 필요합니다. 이 부분은 gcc 구현(libstdc++)을 사용합니다.

MinGW에서는 pthreads-win32 (http://www.sourceware.org/pthreads-win32/) 라이브러리를 함께 배포합니다. pthreads-win32는 pthread 함수들을 윈도 API로 구현한 래퍼 라이브러리로, pthread로 작성된 프로그램을 손쉽게 윈도로 포팅하도록 돕습니다. 다만 pthreads-win32는 LGPL 라이선스라는 점과, pthread POSIX에서 명시되지 않은 부분에 대한 호환성 이슈가 있다는 점에 주의할 필요는 있습니다.

MinGW의 target triple은 `i686-pc-mingw32`입니다. i686은 아키텍처(32비트), pc는 벤더(큰 의미는 없습니다), mingw32는 OS/플랫폼을 의미합니다.

한가지 주의해야 할 부분은 MinGW는 x86_64 아키텍처를 지원하지 않기 때문에 MinGW에서 64비트 바이너리를 만들 수는 없다는 점입니다. (아래에서 설명할 MinGW-w64가 64비트를 지원합니다.)

MinGW는 현재 sf.net git 저장소 (http://sourceforge.net/p/mingw/_list/git)에서 관리됩니다. git 구조가 꽤 난해한데, 가장 중요한 런타임/헤더 코드는 mingw-org-wsl (http://sourceforge.net/p/mingw/mingw-org-wsl/)에 있습니다.

MSYS

MinGW 프로젝트에서는 헤더/런타임 이외에도 MSYS (http://www.mingw.org/wiki/MSYS)를 개발합니다. MSYS는 bash나 make 등의 유닉스 중요 프로그램이 윈도에서도 동작하기 위한 최소한의 환경을 지원하며, Cygwin 초기 버전을 fork하여 개발되었습니다.

가령 MSYS에서는 유닉스와 비슷하게 동작하는 파일시스템을 지원하며, `/usr`나 `/home` 같은 유닉스 path를 이해합니다. MSYS의 `make`는 이 유닉스 path를 기반으로 `Makefile`을 수행해줍니다. 다만 MSYS는 cygwin과 같은 완전한 Unix 레이어를 지원하는 것은 아니며, 개발에 필요할 정도의 최소한의 환경만을 제공합니다.

최근에는 Cygwin 최신 버전을 기반으로 하는 MSYS2 (https://msys2.github.io/)가 관리되고 있습니다. pacman이라는 Arch linux 패키지 매니저를 msys2로 포팅해 두었고, 덕택에 쉽게 필요 패키지를 설치할 수 있습니다.

MinGW-w64

MinGW-w64 (http://mingw-w64.sourceforge.net/)는 MinGW의 64비트 fork가 아닙니다! MinGW-w64는 MinGW와는 코드를 공유하지 않습니다.

과거 OneVision이라는 회사에서는 Objective-C 코드를 64비트 윈도 환경에 포팅하고 싶었지만, MinGW가 64비트 윈도를 지원하지 못하는 문제가 있었습니다. 이에 OneVision에서는 윈도 gcc 64비트에 필요한 부분을 직접 바닥부터 제작하였다고 합니다. 하지만, 이렇게 제작된 코드를

MinGW 측에 제출하자 MinGW 측에서는 이 구현체가 clean room 구현인지 아닌지 확신할 수 없어(당시 회사의 사정으로 구현체에 대해 많은 정보를 공개할 수 없었다고 합니다) 패치를 거부하였고 (http://article.gmane.org/gmane.comp.gnu.mingw.devel/3390), 이에 제작자는 sourceforge에 MinGW-w64라는 이름으로 별도 프로젝트를 시작하게 됩니다.

이러한 이유로, MinGW-w64는 MinGW과 밀접하게 관련되어 있지만 전혀 별개의 구현체입니다. MinGW-w64라는 이름과 달리 32비트 환경도 함께 지원하고요. 오히려 MinGW-w64 쪽이 MinGW보다 32비트 지원이 더 좋습니다! (Qt는 MinGW-w64를 사용하고 있는데, 선택 기준 문서 (http://qt-project.org/wiki/MinGW-64-bit)에서 자세한 비교를 볼 수 있습니다.)

MinGW-w64의 64비트 triple은 `x86_64-w64-mingw32` 입니다. `x86_64`는 CPU 정보, `w64`는 벤더가 MinGW-w64라는 의미입니다. OS/플랫폼에는 MinGW과 동일하게 `mingw32`가 쓰입니다. MinGW-w64의 32비트 triple은 `i686-w64-mingw32` 입니다. 점점 헷갈려 오는데, 보통은 MinGW인가 MinGW-w64인가를 구별할 필요는 없으며 맨 앞이 `i686`인가 `x86_64`인가만 보면 됩니다. (LLVM에서는 둘을 구별하지 않고 `*-pc-windows-gnu`라는 triple로 통합하여 씁니다.)

MinGW 3.x와 비슷하게, MinGW-w64의 헤더 파일들은 퍼블릭 도메인으로 배포됩니다. 하지만 MinGW-w64에서 사용하는 DirectX 헤더는 Wine (http://www.winehq.org/) 구현체를 가져온 것이며 LGPL을 따릅니다 (http://sourceforge.net/apps/trac/mingw-w64/browser/trunk/mingw-w64-headers/direct-x/readme.txt?rev=5243). 비슷하게 DDK 부분 (http://sourceforge.net/apps/trac/mingw-w64/browser/trunk/mingw-w64-headers/ddk/readme.txt?rev=5243)은 ReactOS (http://www.reactos.org/)에서 가져왔고 이중 일부분은 LGPL로 배포됩니다.

한편 MinGW-w64에서는 pthreads-win32를 대체하는 winpthreads를 직접 개발합니다. 이 라이브러리는 pthreads-win32에 비교하여 다음과 같은 장점을 가집니다.

- 64비트 환경에서 더 나은 지원을 제공합니다.
- pthreads-win32는 `pthread_t`가 값 타입이 아니라 struct 타입이며, 이것은 값 타입으로 가정하여 작성된 프로그램에서 문제가 될 수 있습니다. (POSIX에서는 해당 타입이 struct여도 되지만, 많은 프로그램이 값 타입을 가정하고 작성되어 현실적인 문제가 있다고 합니다.) winpthreads는 `pthread_t`를 값 타입으로 사용합니다.
- pthreads-win32는 LGPL이지만 winpthreads는 MIT 라이선스입니다.

다만 아직 몇몇 부분에서는 pthreads-win32에 비교하여 성능 저하가 있을 수 있습니다.

MinGW-w64는 설치하는 것도 그리 쉬운 편이 아닙니다. MinGW에서는 '공식 바이너리'를 하나만 제공하기 때문에 별다른 고민이 없지만 MinGW-w64의 다운로드 페이지 (http://mingw-w64.sourceforge.net/download.php)는 무엇을 선택해야 하는지 잘 안 보입니다. 선택지 중에 mingw-builds가 보이는데, mingw-builds는 MinGW-w64와 함께 작업하고 있기 때문에 (http://permalink.gmane.org/gmane.comp.gnu.mingw.w64.general/8484) 여기에서 제공하는 인스톨러를 사용하는 것이 가장 좋습니다. 혹은 MSYS2 (https://msys2.github.io/)를 설치한 다음 `mingw-w64-{i686,x86_64}-toolchain` 패키지를 설치하는 것도 하나의 방법입니다.

mingw-builds (http://sourceforge.net/projects/mingwbuilds/) 인스톨러를 받아 실행해보면 세 가지를 선택해야 합니다. 먼저 32bit/64bit를 선택해야 합니다. 다음은 스레드 모델을 골라야 한다는데 pthread와 win32가 있습니다. 그 다음은 exception handling도 하나 골라야 합니다. sjlj에

dwarf에 seh라는 알 수 없는 단어들이 등장합니다.

이 옵션들은 ABI 호환성과 성능에 직접적인 영향을 주기 때문에 정확히 선택해야 합니다.

Thread Model

MinGW-w64에서는 win32와 posix라는 두 스레드 API를 지원합니다. win32는 윈도 API의 스레드 API를 의미하며(여기서 32는 32/64비트와는 관계없습니다), posix는 winpthreads로 구현된 pthread API를 의미합니다.

물론 이 두 API는 동시에 사용할 수 있습니다. win32 옵션을 선택해도 winpthreads를 쓸 수 있고요. 그렇다면 왜 둘 중 하나를 골라야 하냐면 C++11 thread (http://en.cppreference.com/w/cpp/thread) 지원 문제 때문입니다.

MinGW-w64에서는 C++ 표준 라이브러리로 libstdc++를 사용합니다. 하지만 libstdc++의 C++11 thread 구현은 pthread 기반으로 작성되어 있습니다. 즉, C++11 thread를 사용하기 위해서는 posix thread 지원이 필요합니다.

Exception Handling

Mingw-w64/gcc에서 사용 가능한 exception handling(EH) 모델에는 SEH, SJLJ, DWARF 세 가지가 있습니다.

SEH(structured exception handling)는 윈도에서 기본적으로 사용하는 EH 방법을 부르는 통칭으로, 32비트와 64비트의 EH는 완전히 다른 구조를 가집니다 (http://blogs.msdn.com/b/oldnewthing/archive/2013/03/20/10403718.aspx).

32비트 EH는 함수 prolog 부분에 추가 정보를 기록하는 방식 (http://www.microsoft.com/msj/0197/Exception/Exception.aspx)인데, 이 방식은 Exception이 발생하지 않는 경우에도 함수 호출마다 오버헤드가 발생하는 단점이 있습니다. 더 큰 문제는 해당 메커니즘에 대한 특허를 볼랜드가 가지고 있을 가능성 (http://wiki.ffii.org/Wine05En)이 있었고, 이런 문제로 gcc와 LLVM에서는 2015년 3월 현재 해당 방식을 구현하지 않았습니다. 해당 특허는 2014년 6월에 만료되었으며, 이후에는 32비트 SEH가 구현될 가능성도 있을 겁니다.

64비트SEH는 특허 이슈가 없으며, 오버헤드도 없습니다! MinGW-w64/gcc과 LLVM은 64비트 EH를 지원합니다. 64비트에서는 SEH가 최선의 선택입니다.

32비트의 경우 SJLJ와 DWARF 두 가지 선택이 있습니다. SJLJ는 setjmp와 longjmp (http://en.wikipedia.org/wiki/Setjmp.h#Exception_handling)를 사용하는 방법으로, 매 함수마다 setjmp를 사용하여 예외 발생시 어떤 일을 할지를 기록합니다. SJLJ는 호환성 문제에서 자유로운데, 다른 라이브러리가 다른 EH 방식을 쓴다고 해도 두 EH 방식이 섞인다고 해서 문제가 발생하지는 않습니다. 하지만 런타임 오버헤드가 추가된다는 문제가 있습니다.

DWARF는 DWARF EH (http://wiki.dwarfstd.org/index.php?title=Exception_Handling)를 사용하는 방식입니다. (DWARF EH는 Linux의 기본 EH 방식입니다.) 런타임 오버헤드가 없지만, 다른 EH 방식과의 호환성 문제가 있습니다. libgcc의 EH 지원 라이브러리를 dll에 정적 링크할 경우 (`-static-libgcc`) dll간에 exception을 넘길 수 없다는 문제도 있습니다. (동적 링크한다면 런타임 의존성에 `libgcc_s_dw2-1.dll`가 추가되기 때문에 약간 더 귀찮아집니다.)

32비트의 경우 DWARF가 성능상으로는 좋지만, 신경써야 할 부분이 늘어납니다. 이 때문에 MinGW-w64에서는 이러한 이슈를 잘 모를 경우 SJLJ를 추천하는 상황입니다. 리눅스의 MinGW-w64 크로스 컴파일러 패키지 역시 대부분 SJLJ를 사용하기 때문에 ABI 문제도 있고요.
